Sharing Equality is Linear

Andrea Condoluci, Beniamino Accattoli, Claudio Sacerdoti Coen

Sharing Equality is Linear

Andrea Condoluci
Department of Computer Science and Engineering
University of Bologna
Italy
andrea.condoluci@unibo.it

Beniamino Accattoli
LIX
Inria & École Polytechnique
France
beniamino.accattoli@inria.fr

Claudio Sacerdoti Coen
Department of Computer Science and Engineering
University of Bologna
Italy
claudio.sacerdoticoen@unibo.it

ABSTRACT

The λ-calculus is a handy formalism to specify the evaluation of higher-order programs. It is not very handy, however, when one interprets the specification as an execution mechanism, because terms can grow exponentially with the number of β-steps. This is why implementations of functional languages and proof assistants always rely on some form of sharing of subterms.

These frameworks however do not only evaluate λ-terms, they also have to compare them for equality. In presence of sharing, one is actually interested in equality of the underlying *unshared* λ-terms. The literature contains algorithms for such a *sharing equality*, that are polynomial in the sizes of the shared terms.

This paper improves the bounds in the literature by presenting the first *linear time* algorithm. As others before us, we are inspired by Paterson and Wegman's algorithm for first-order unification, itself based on representing terms with sharing as DAGs, and sharing equality as bisimulation of DAGs. Beyond the improved complexity, a distinguishing point of our work is a dissection of the involved concepts. In particular, we show that the algorithm computes the smallest bisimulation between the given DAGs, if any.

KEYWORDS

lambda-calculus, sharing, alpha-equivalence, bisimulation

ACM Reference Format:
Andrea Condoluci, Beniamino Accattoli, and Claudio Sacerdoti Coen. 2019. Sharing Equality is Linear. In *Principles and Practice of Programming Languages 2019 (PPDP '19), October 7–9, 2019, Porto, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3354166.3354174

1 INTRODUCTION

Origin and Downfall of the Problem

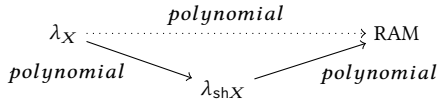
For as strange as it may sound, the λ-calculus is not a good setting for evaluating and representing higher-order programs. It is an excellent specification framework, but—it is simply a matter of fact—no tool based on the λ-calculus implements it as it is.

Reasonable evaluation and sharing. Fix a dialect λ_X of the λ-calculus with a deterministic evaluation strategy $\to_X$, and note $\mathtt{nf}_X(t)$ the normal form of t with respect to $\to_X$. If the λ-calculus were a reasonable execution model then one would at least expect that mechanizing an evaluation sequence $t \to_X^n \mathtt{nf}_X(t)$ on random access machines (RAM) would have a cost polynomial in the size of t and in the number n of β-steps. In this way a program of λ_X evaluating in a polynomial number of steps can indeed be considered as having polynomial cost.

Unfortunately, this is not the case, at least not literally. The problem is called *size explosion*: there are families of terms whose size grows exponentially with the number of evaluation steps, obtained by nesting duplications one inside the other—simply writing down the result $\mathtt{nf}_X(t)$ may then require cost exponential in n.

In many cases sharing is the cure because size explosion is caused by an unnecessary duplications of subterms, that can be avoided if such subterms are instead shared, and evaluation is modified accordingly.

The idea is to introduce an intermediate setting $\lambda_{\mathsf{sh}X}$ where λ_X is refined with sharing (we are vague about sharing on purpose) and evaluation in λ_X is simulated by some refinement $\to_{\mathsf{sh}X}$ of $\to_X$. A term with sharing t represents the ordinary term $t\!\downarrow$ obtained by unfolding the sharing in t—the key point is that t can be exponentially smaller than $t\!\downarrow$. Evaluation in $\lambda_{\mathsf{sh}X}$ produces a shared normal form $\mathtt{nf}_{\mathsf{sh}X}(t)$ that is a compact representation of the ordinary result, that is, such that $\mathtt{nf}_{\mathsf{sh}X}(t)\!\downarrow \;= \mathtt{nf}_X(t\!\downarrow)$. The situation can then be refined as in the following diagram:



Let us explain it. One says that λ_X is *reasonably implementable* if both the simulation of λ_X in $\lambda_{\mathsf{sh}X}$ up to sharing and the mechanization of $\lambda_{\mathsf{sh}X}$ can be done in time polynomial in the size of the initial term t and of the number n of β-steps. If λ_X is reasonably implementable then it is possible to reason about it as if it were not suffering of size explosion. The main consequence of such a schema is that the number of β-steps in λ_X then becomes a reasonable complexity measure—essentially the complexity class P defined in λ_X coincides with the one defined by RAM or Turing machines.

The first result in this area appeared only in the nineties and for a special case—Blelloch and Greiner showed that weak (that is, not under abstraction) call-by-value evaluation is reasonably implementable [10]. The strong case, where reduction is allowed everywhere, has received a positive answer only in 2014, when Accattoli and Dal Lago have shown that leftmost-outermost evaluation is reasonably implementable [6].

PPDP '19, October 7–9, 2019, Porto, Portugal

ACM ISBN 978-1-4503-7249-7/19/10…$15.00
https://doi.org/10.1145/3354166.3354174

Reasonable conversion and sharing. Some higher-order settings need more than evaluation of a single term. They often also have to check whether two terms t and s are X-convertible—for instance to implement the equality predicate, as in OCaml, or for type checking in settings using dependent types, typically in Coq. These frameworks usually rely on a set of folklore and ad-hoc heuristics for conversion, that quickly solve many frequent special cases. In the general case, however, the only known algorithm is to first evaluate t and s to their normal forms $\mathtt{nf}_X(t)$ and $\mathtt{nf}_X(s)$ and then check $\mathtt{nf}_X(t)$ and $\mathtt{nf}_X(s)$ for equality—actually, for α-equivalence because terms in the λ-calculus are identified up to α. One can then say that conversion in λ_X is *reasonable* if checking $\mathtt{nf}_X(t) =_\alpha \mathtt{nf}_X(s)$ can be done in time polynomial in the sizes of t and s and in the number of β steps to evaluate them.

Sharing is the cure for size explosion during evaluation, but what about conversion? Size explosion forces reasonable evaluations to produce shared results. Equality in λ_X unfortunately does not trivially reduce to equality in $\lambda_{\mathtt{sh}X}$, because a single term admits many different shared representations in general. Therefore, one needs to be able to test *sharing equality*, that is to decide whether $t{\downarrow} =_\alpha s{\downarrow}$ given two shared terms t and s.

For conversion to be reasonable, sharing equality has to be testable in time polynomial in the sizes of t and s. The obvious algorithm that extracts the unfoldings $t{\downarrow}$ and $s{\downarrow}$ and then checks α-equivalence is of course too naïve, because computing the unfolding is exponential. The tricky point therefore is that sharing equality has to be checked without unfolding the sharing.

In these terms, the question has first been addressed by Accattoli and Dal Lago in [5], where they provide a quadratic algorithm for sharing equality. Consequently, conversion is reasonable.

A closer look to the costs. Once established that strong evaluation and conversion are both reasonable it is natural to wonder how efficiently can they be implemented. Accattoli and Sacerdoti Coen in [3] essentially show that strong evaluation can be implemented within a bilinear overhead, *i.e.* with overhead linear in the size of the initial term and in the number of β-steps. Their technique has then been simplified by Accattoli and Guerrieri in [4]. Both works actually address *open* evaluation, which is a bit simpler than strong evaluation—the moral however is that evaluation is bilinear. Consequently, the size of the computed result is bilinear.

The bottleneck for conversion then seemed to be Accattoli and Dal Lago's quadratic algorithm for sharing equality. The literature actually contains also other algorithms, studied with different motivations or for slightly different problems, discussed among related works in the next section. None of these algorithms however matches the complexity of evaluation.

In this paper we provide the first algorithm for sharing equality that is linear in the size of the shared terms, improving over the literature. Therefore, the complexity of sharing equality matches the one of evaluation, providing a combined bilinear algorithm for conversion, that is the real motivation behind this work.

Computing Sharing Equality

Sharing as DAGs. Sharing can be added to λ-terms in different forms. In this paper we adopt a graphical approach. Roughly, a λ-term can be seen as a (sort of) directed tree whose root is the topmost constructor and whose leaves are the (free) variables. A λ-term with sharing is more generally a Directed Acyclic Graph (DAG). Sharing of a subterm t is then the fact that the root node n of t has more than one parent.

This type of sharing is usually called *horizontal* or subterm sharing, and it is essentially the same sharing as in calculi with explicit substitution, environment-based abstract machines, or linear logic—the details are different but all these approaches provide different incarnations of the same notion of sharing. Other types of sharing include so-called *vertical sharing* (μ, `letrec`), twisted sharing [11], and *sharing graphs* [24]. The latter provide a much deeper form of sharing than our DAGs, and are required by Lamping's algorithm for optimal reduction. To our knowledge, sharing equality for sharing graphs has never been studied—it is not even known whether it is reasonable.

Sharing equality as bisimilarity. When λ-terms with sharing are represented as DAGs, a natural way of checking sharing equality is to test DAGs for bisimilarity. Careful here: the transition system under study is the one given by the directed edges of the DAG, and not the one given by β-reduction steps, as in applicative bisimilarity—our DAGs may have β-redexes but we do not reduce them in this paper, that is an orthogonal issue, namely evaluation. Essentially, two DAGs represent the same unfolded λ-term if they have the same structural paths, just collapsed differently.

To be precise, sharing equality is based on what we call *sharing equivalences*, that are bisimulations plus some additional requirements about free variables and the requirement that they are equivalence relations.

Binders, cycles, and domination. A key point of our problem is the presence of binders, *i.e.* abstractions, and the fact that equality on usual λ-terms is α-equivalence. Graphically, it is standard to see abstractions as getting a backward edge from the variable they bind—a way of representing scopes that dates back to Bourbaki in *Eléments de Théorie des Ensembles*, but also supported by the strong relationship between λ-calculus and linear logic proof nets.

In this approach, binders introduce a form of cycle in the λ-graph: while two free variables are bisimilar only if they coincide, two bound variables are bisimilar only when also their binders are bisimilar, suggesting that λ-terms with sharing are, as directed graphs, structurally closer to deterministic finite automata (DFA), that may have cycles, than to DAGs. The problem with cycles is that in general bisimilarity of DAGs cannot be checked in linear time—Hopcroft and Karp's algorithm [23], the best one, is only pseudo-linear, that is, with an inverse Ackermann factor.

Technically speaking, the cycles induced by binders are not actual cycles: unlike usual downward edges, backwards edges do not point to subterms of a node, but are merely a graphical representation of *scopes*. They are indeed characterized by a structural property called *domination*—exploring the DAG from the root, one necessarily visits the binder before the bound variable. Domination turns out to be one of the key ingredients for a linear algorithm in presence of binders.

Previous work. Sharing equality bears similarities with unification, which are discussed in the next section. For what concerns

sharing equality itself, in the literature there are only two algorithms explicitly addressing it. First, the already cited quadratic one by Accattoli and Dal Lago. Second, a $O(n \log n)$ algorithm by Grabmayer and Rochel [22] (where n is the sum of the sizes of the shared terms to compare, and the input of the algorithm is a graph), obtained by a reduction to equivalence of DFAs and treating the more general case of λ-terms with `letrec`.

Contributions: a theory and a 2-levels linear algorithm. This paper is divided in two parts. The first part develops a re-usable, self-contained, and clean theory of sharing equality, independent of the algorithm that computes it. Some of its concepts are implicitly used by other authors, but never emerged from the collective unconscious before (*propagated queries* in particular)—others instead are new. A key point is that we bypass the use of α-equivalence by relating sharing equalities on DAGs with λ-terms represented in a locally nameless way [17]. In such an approach, bound names are represented using de Bruijn indices, while free variables are represented using names—thus α-equivalence collapses on equality. The theory culminates with the sharing equality theorem, which connects equality of λ-terms and sharing equivalences on shared λ-terms, under suitable conditions.

The second part studies a linear algorithm for sharing equality by adapting Paterson and Wegman's (shortened to PW) linear algorithm for first-order unification [28] to λ-terms with sharing. Our algorithm is actually composed by a 2-levels, modular approach, pushing further the modularity suggested—but not implemented—in the nominal unification study by Calvès & Fernández in [15]:

- *Blind check*: a reformulation of PW from which we removed the management of meta-variables for unification. It is used as a first-order test on λ-terms with sharing, to check that the unfolded terms have the same skeleton, ignoring variables.
- *Variables check*: a straightforward algorithm executed after the previous one, testing α-equivalence by checking that bisimilar bound variables have bisimilar binders and that two different free variables are never equated.

The decomposition plus the correctness and the completeness of the checks crucially rely on the theory developed in the first part.

The value of the paper. It is delicate to explain the value of our work. Three contributions are clear: 1) the improved complexity of the problem, 2) the consequent downfall on the complexity of β-conversion, and 3) the isolation of a theory of sharing equality. At the same time, however, our algorithm looks as an easy adaptation of PW, and binders do not seem to play much of a role. Let us then draw attention to the following points:

- *Identification of the problem*: the literature presents similar studies and techniques, and yet we are the first to formulate and study the problem *per se* (unification is different, and it is usually not formulated on terms with sharing), directly (*i.e.* without reducing it to DFAs, like in Grabmayer and Rochel), and with a fine-grained look at the complexity (Accattoli and Dal Lago only tried not to be exponential).
- *The role of binders*: the fact that binders can be treated straightforwardly is—we believe—an insight and not a weakness of our work. Essentially, domination allows to reduce sharing equality in presence of binders to the Blind check, under key assumptions on the context in which terms are tested (see *queries*, Sect. 4).
- *Minimality.* The set of shared representations of an ordinary λ-term t is a lattice: the bottom element is t itself, the top element is the (always existing) maximally sharing of t, and for any two terms with sharing there exist *inf* and *sup*. Essentially, Accattoli & Dal Lago and Grabmayer & Rochel address sharing equality by computing the top elements of the lattices of the two λ-terms with sharing, and then comparing them for α-equivalence. We show that our Blind check—and morally every PW-based algorithm—computes the *sup* of t and s, that is, the term having all and only the sharing in t or s, that is the smallest sharing equivalence between the two DAGs. This insight, first pointed out in PW's original paper to characterize most general unifiers, is a prominent concept in our theory of sharing equality as well.
- *Proofs, invariants, and detailed development.* We provide detailed correctness, completeness, and linearity proofs, plus a detailed treatment of the relationship between equality on locally nameless λ-terms and sharing equivalences on λ-graphs. Our work is therefore self-contained, but for the fact that most of the theorems and their proofs have been omitted from the body for lack of space. The complete technical development can be found in the accompanying extended version of this paper [18].
- *Concrete implementation.* We implemented our algorithm[1] and verified experimentally its linear time complexity. However, let us stress that despite providing an implementation our aim is mainly theoretical. Namely, we are interested in showing that sharing equality is linear (to obtain that conversion is bilinear) and not only pseudo-linear, even though other algorithms with super-linear asymptotic complexity may perform better in practice.

2 RELATED PROBLEMS AND TECHNICAL CHOICES

We suggest the reader to skip this section at a first reading.

Related problems. There are various problems that are closely related to sharing equality, and that are also treated with bisimilarity-based algorithms. Let us list similarities and differences:

- *First-order unification.* On the one hand the problem is more general, because unification roughly allows to substitute variables with terms not present in the original DAGs, while in sharing equality this is not possible. On the other hand, the problem is less general, because it does not allow binders and does not test α-equivalence. There are basically two linear algorithm for first-order unification, Paterson and Wegman's (PW) [28] and Martelli and Montanari's (MM) [27]. Both rely on sharing to be linear. PW even takes terms with sharing as inputs, while MM deals with sharing in a less direct way, except in its less known variant [26] that takes in input terms shared using the Boyer-Moore technique [12].
- *Nominal unification* is unification up to α-equivalence (but not up to β or η equivalence) of λ-calculi extended with name

[1] The code is available on http://www.cs.unibo.it/~sacerdot/sharing_equivalence.tgz.

swapping, in the nominal tradition. It has first studied by Urban, Pitts, and Gabbay [32] and efficient algorithms are due to two groups, Calvès & Fernández and Levy & Villaret, adapting PW and MM form first-order unification. It is very close to sharing equality, but the known best algorithms [15, 25] are only quadratic. See [14] for a unifying presentation.
- *Pattern unification.* Miller's pattern unification can also be stripped down to test sharing equality. Qian presents a PW-inspired algorithm, claiming linear complexity [29], that seems to work only on unshared terms. We say *claiming* because the algorithm is very involved and the proofs are far from being clear. Moreover, according to Levy and Villaret in [25]: *it is really difficult to obtain a practical algorithm from the proof described in [29].* We believe that is fair to say that Qian's work is hermetic (please try to read it!).
- *Nominal matching.* Calvès & Fernández in [16] present an algorithm for nominal matching (a special case of unification) that is linear, but *only* on unshared input terms.
- *Equivalence of DFA.* Automata do not have binders, and yet they are structurally more general than λ-terms with sharing, since they allow arbitrary directed cycles, not necessarily dominated. As already pointed out, the best equivalence algorithm is only pseudo-linear [23].

Algorithms for α-equivalence extended with further principles (*e.g.* permutations of let expressions), but not up to sharing unfolding, is studied by Schmidt-Schauß, Rau, and Sabel in [31].

Hash-consing. Hash-consing [19, 21] is a technique to share purely functional data, traditionally realised through a hash table [7]. It is a eager approach to conversion of λ-terms, in which one keeps trace in a huge table of all the pairs of convertible terms previously encountered. Our approach is somewhat dual or lazy, as our technique does not record previous tests, only the current equality problem is analysed. With hash consing, to check that two terms are sharing equivalent it suffices to perform a lookup in the hash table, but first the terms have to be hash-consed (*i.e.* maximally shared), which requires quasilinear running times.

Technical choices. Our algorithm requires λ-terms to be represented as graphs. This choice is fair, since abstract machines with global environments such as those described by Accattoli and Barras in [2] do manipulate similar representations, and thus produce λ-graphs to be compared for sharing equality—moreover, it is essentially the same representation induced by the translation of λ-calculus into linear logic proof nets [1, 30] up to explicit sharing nodes—namely ?-links—and explicit boxes. In this paper we do not consider explicit sharing nodes (that in λ-calculus syntax correspond to have sharing only on variables), but one can translate in linear time between that representation to ours (and viceversa) by collapsing these *variables-as-sharing* on their child if they are the child of some other node. Our results could be adapted to this other approach, but at the price of more technical definitions.

On the other hand, abstract machines with *local* environments (*e.g.* Krivine asbtract machine), that typically rely on de Bruijn indices, produce different representations where every subterm is paired with an environment, representing a delayed substitution. Those outputs cannot be directly compared for sharing equality, they first have to be translated to a global environment representation—the study of such a translation is future work.

3 PRELIMINARIES

In this section we introduce λ-graphs, the representation of λ-terms with sharing that we consider in this paper. First of all we introduce the usual λ-calculus without sharing.

λ-terms and equality. The syntax of λ-terms is usually defined as follows:

$$\text{(Named) Terms} \qquad t, s \quad ::= \quad x \mid \lambda x.t \mid t\ s$$

This representation of λ-terms is called *named*, since variables and abstractions bear names. Equality on named terms is not just syntactic equality, but α-equality: terms are identified up to the renaming of bound variables, because for example the named terms $\lambda x.\lambda y.x$ and $\lambda y.\lambda x.y$ should denote the same λ-term.

This is the standard representation of λ-terms, but reasoning in presence of names is cumbersome, especially names for bound variables as they force the introduction of α-equivalence classes. Moreover nodes naturally bear no names, thus we prefer to avoid assigning arbitrary variable names when performing the readback of a node to a λ-term.

On *closed* λ-terms (terms with only bound variables), this suggests a *nameless* approach: by using *de Bruijn indices*, variables simply consist of an *index*, a natural number indicating the number of abstractions that occur between the abstraction to which the variable is bound and that variable occurrence. For example, the named term $\lambda x.\lambda y.x$ and the nameless term $\lambda\lambda\underline{1}$ denote the same λ-term. The nameless representation can be extended to *open* λ-terms, but it adds complications because for example different occurrences of the same free variable unnaturally have different indices.

Since we are forced to work with open terms (because proof assistants require them) the most comfortable representation and the one we adopt in our proofs is the *locally nameless* representation (see Charguéraud [17] for a thorough discussion). This representation combines named and nameless, by using de Bruijn indices for bound variables (thus avoiding the need for α-equivalence), and names for free variables. This representation of λ-terms is the most faithful to our definition of λ-graphs: as we shall see, to compare two bound variable nodes one compares their binders, but to compare two free variable nodes one uses their identifier, which in implementations is usually their memory address or a user-supplied string.

The syntax of locally nameless (l.n.) terms is:

$$\text{(l.n.) Terms} \qquad t, s \quad ::= \quad \underline{i} \mid \overline{a} \mid t\ s \mid \lambda t \quad (i \in \mathbb{N},\ a \in \mathbb{A})$$

where $\underline{i}$ denotes a bound variable of de Bruijn index i ($\mathbb{N}$ is the set of natural numbers), and $\overline{a}$ denotes a free variable of name a ($\mathbb{A}$ is a set of *atoms*).

In the rest of the paper, we switch seamlessly between different representations of λ-terms (without sharing): we use named terms in examples, as they are more human-friendly, but locally nameless terms in the technical parts, since much more elegant and short proofs can be obtained by avoiding to reason explicitly on bound names and α-equivalence.

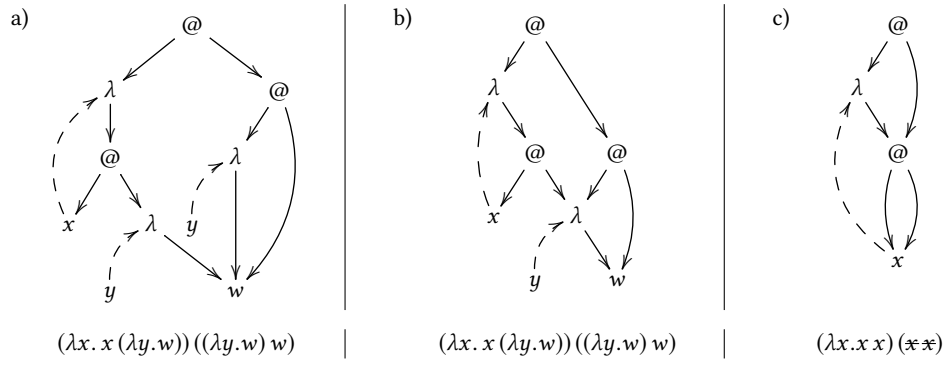


Figure 1: *a)* λ-term as a λ-tree, without sharing; *b)* λ-graph with sharing (same term of a); *c)* λ-graph breaking domination.

Terms as graphs, informally. Graphically, λ-terms can be seen as syntax trees—please have a look at the example in Fig. 1(a). Note that in all Fig. 1 we provide names to bound variables (both in the graphs and in the terms below) only to make the examples more understandable; as already mentioned, our upcoming notion of λ-graph follows the locally nameless convention (as in Fig. 2).

As seen in the figure, we apply two tweaks relative to variable nodes:

- *Binding edges*: bound variable nodes have a binding (dashed) edge towards the abstraction node that binds them;
- *Variables merging*: all the nodes corresponding to the occurrences of a same variable are merged together, like the three occurrences of w in the example. In this way, the equality of free variable nodes is just the physical equality of nodes.

Sharing is realized by simply allowing all nodes to have more than one parent, as for instance the abstraction on y in Fig. 1(b) — note that sharing can happen inside abstractions, *e.g.* $\lambda y.w$ is shared under the abstraction on x.

Our notion of λ-terms with sharing is given in Theorem 7. It is built in in two steps: we first introduce pre–λ-graphs, and later define the required structural properties that make them λ-graphs.

Definition 1 (pre–λ-graphs). *A* pre–λ-graph *is a directed graph with three kind of nodes:*

- Application*: an application node is labelled with* App *and has exactly two children, called* left *(↙) and* right *(↘). We write* App(n, m) *for a node labelled by* App *whose left child is n and whose right child is m.*
- Abstraction*: an abstraction node is labelled with* Abs *and has exactly one child, called its* body *(↓). We write* Abs(n) *for a node labelled by* Abs *with body n. We denote by l a generic abstraction node.*
- Variable*: a variable node is labelled with* Var*, and may be* free *or* bound*:*
  - *A* free *variable node has no children, and is denoted by* Var()*. We assume a function* name$(\cdot)$ *that assigns to every free variable node n an atom* name$(n) \in \mathbb{A}$ *that uniquely identifies it*[2]*: this identifier is going to be used later, when defining the readback of nodes to λ-terms (cf. Theorem 9).*
  - *A bound* variable *node has exactly one child, called its* binder *(↻). We write* Var(l) *for a node labelled by* Var *with binder l. The corresponding binding edge is represented with a dashed line.*

[2]We also assume that the set of atoms $\mathbb{A}$ is such that the equality of atoms can be tested in constant time.

Domination. Not every pre–λ-graph represents a λ-graph. For instance, the graph in Fig. 1(c) does not, because the bound variable x is visible outside the scope of its abstraction, since there is a path to x from the application *above* the abstraction that does not pass through the abstraction itself. One would say that such a graph represents $(\lambda x.xx)(xx)$, but the variable x in xx and the one in $\lambda x.xx =_\alpha \lambda y.yy$ cannot be the same.

It is well-known that scopes corresponding to λ-terms are characterized by a property borrowed from control-flow graphs called *domination* (also called *unique binding* by Wadsworth [33], and checkable in linear time [8, 13, 20]): given two nodes n and m of a graph G, we say that n *dominates* m when every path from a root of G to m crosses n. To define this property formally, we first need to define paths in a pre–λ-graph.

Notations for paths. Paths are a crucial concept, needed both to define the readback to λ-terms and to state formally the properties of λ-graphs of being acyclic and dominated.

A path in a graph is determined by a start node together with a *trace, i.e.* a sequence of directions. The allowed directions are "↙", "↓", and "↘": we are not going to consider paths that use binding edges "↻", because these edges simply denote scoping and do not actually point to subterms of (the readback of) a node.

Definition 2 (Paths, traces). *We define* traces *as finite sequences of directions:*

$$\textsc{Directions} \quad d ::= \swarrow \mid \downarrow \mid \searrow$$
$$\textsc{Traces} \quad \tau ::= \epsilon \mid \tau \cdot d$$

Let n, m be nodes of a pre–λ-graph, and τ be a trace. We define inductively the judgement "$n \overset{\tau}{\rightsquigarrow} m$", which reads "path from n to m (of trace τ)":

- Empty*: $n \overset{\epsilon}{\rightsquigarrow} n$.*
- Abstraction*: if $n \overset{\tau}{\rightsquigarrow}$* Abs$(m)$*, then $n \overset{\tau\cdot\downarrow}{\rightsquigarrow} m$.*
- Application*: if $n \overset{\tau}{\rightsquigarrow}$* App$(m_1, m_2)$*, then $n \overset{\tau\cdot\swarrow}{\rightsquigarrow} m_1$ and $n \overset{\tau\cdot\searrow}{\rightsquigarrow} m_2$.*

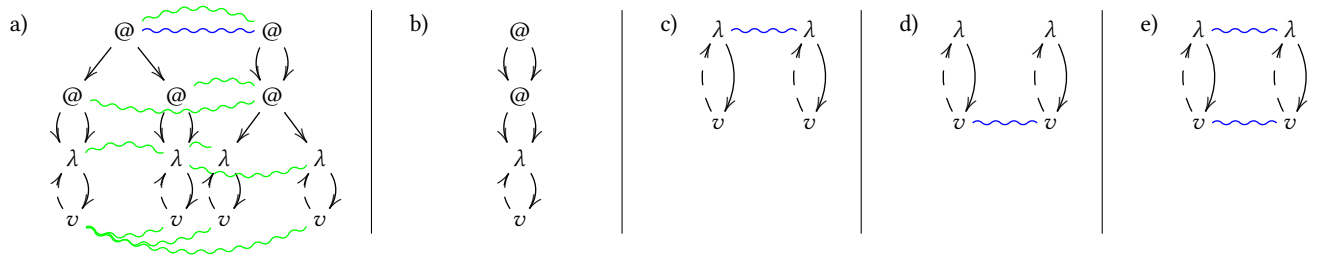


Figure 2: Examples of sharing equivalences and queries.

We just write $n \overset{\tau}{\rightsquigarrow}$ if $n \overset{\tau}{\rightsquigarrow} m$ for some node m when the endpoint m is not relevant.

To state formally the requirements that make a pre–λ-graph a λ-graph, we need two additional concepts: the roots of a pre–λ-graph, and paths crossing a node.

Root nodes. pre–λ-graphs (and later λ-graphs) may have various root nodes. What is maybe less expected, is that these roots may share some parts of the graph. Consider Fig. 1(b), and imagine to remove the root and its edges: the outcome is still a perfectly legal pre–λ-graph. We admit these configurations because they actually arise naturally in implementations, especially of proof assistants.

Definition 3 (Roots). *Let r be a node of a pre–λ-graph G. r is a root if and only if the only path with endnode r has empty trace.*

Definition 4 (Path Crossing a Node). *Let n, m be nodes of a pre–λ-graph, and τ a trace such that $n \overset{\tau}{\rightsquigarrow}$. We define inductively the judgment "$n \overset{\tau}{\rightsquigarrow}$ crosses m":*

- *if $n \overset{\tau}{\rightsquigarrow} m$, then $n \overset{\tau}{\rightsquigarrow}$ crosses m*
- *if $n \overset{\tau}{\rightsquigarrow}$ crosses m and $n \overset{\tau \cdot d}{\rightsquigarrow}$, then $n \overset{\tau \cdot d}{\rightsquigarrow}$ crosses m.*

Definition 5 (Dominated pre–λ-graph). *Let G be a pre–λ-graph, and n, m be nodes of G: we say that m dominates n when every path $r \overset{\tau}{\rightsquigarrow} n$ from a root node r crosses m.*

We say that G is dominated *when each bound variable node* $\mathtt{Var}(l)$ *is dominatedd by its binder l.*

Definition 6 (Acyclic pre–λ-graph). *We say that a pre–λ-graph G is* acyclic *when for every node n in G and every trace τ, $n \overset{\tau}{\rightsquigarrow} n$ if and only if $\tau = \epsilon$.*

Our precise definition of λ-terms with sharing follows:

Definition 7 (λ-graphs). *A pre–λ-graph G is a* λ-graph *if it satisfies the following additional structural properties:*

- Finite: *G has a finite number of nodes.*
- Acyclic*: if binding edges are ignored, G is a DAG (Def. 6).*
- Dominated*: bound variable nodes are dominated by their binder (Def. 5).*

Readback. The sharing in a λ-graph can be unfolded by duplicating shared sub-graphs, obtaining a λ-tree. We prefer however to adopt another approach. We define a readback procedure associating a λ-term $[\![r]\!]$ (without sharing) to each root node r of the λ-graph, in such a way that shared sub-graphs simply appear multiple times. However, since we use de Bruijn indices for bound variables, any node by itself does not uniquely identify a λ-term: in fact, its readback depends on the path through which it is reached from a root, also known as its *access path* [9]. That path determines the abstraction nodes encountered, and thus the indices to assign to bound variable nodes. We define formally $\mathtt{index}(l \mid n \overset{\tau}{\rightsquigarrow})$, the index of an abstraction node l according to a path $n \overset{\tau}{\rightsquigarrow}$ crossing l (recall that l, l' denote abstraction nodes):

Definition 8 ($\mathtt{index}(\cdot \mid \cdot)$). *Let n, l be nodes of a λ-graph G, and $n \overset{\tau}{\rightsquigarrow}$ a path crossing l. We define* $\mathtt{index}(l \mid n \overset{\tau}{\rightsquigarrow})$ *by structural induction on the derivation of the judgement "$n \overset{\tau}{\rightsquigarrow}$ crosses l":*

$$\mathtt{index}(l \mid n \overset{\tau}{\rightsquigarrow} l) := 0$$

$$\mathtt{index}(l \mid n \overset{\tau \cdot d}{\rightsquigarrow} l') := \mathtt{index}(l \mid n \overset{\tau}{\rightsquigarrow}) + 1 \qquad \text{if } l \neq l'$$

$$\mathtt{index}(l \mid n \overset{\tau \cdot d}{\rightsquigarrow} m) := \mathtt{index}(l \mid n \overset{\tau}{\rightsquigarrow}) \qquad \text{otherwise.}$$

Definition 9 (Readback to λ-terms $[\![\cdot]\!]$). *Let G be a λ-graph. For every root r and path $r \overset{\tau}{\rightsquigarrow} n$, we define the readback $[\![r \overset{\tau}{\rightsquigarrow} n]\!]$ of n relative to the access path $r \overset{\tau}{\rightsquigarrow} n$, by cases on n:*

(1) $[\![r \overset{\tau}{\rightsquigarrow} \mathtt{Var}(l)]\!] := \underline{i}$ *where* $i := \mathtt{index}(l \mid r \overset{\tau}{\rightsquigarrow})$.
(2) $[\![r \overset{\tau}{\rightsquigarrow} \mathtt{Var}()]\!] := \overline{a}$ *where* $a := \mathtt{name}(n)$.
(3) $[\![r \overset{\tau}{\rightsquigarrow} \mathtt{Abs}(m)]\!] := \lambda\, [\![r \overset{\tau \cdot \downarrow}{\rightsquigarrow} m]\!]$.
(4) $[\![r \overset{\tau}{\rightsquigarrow} \mathtt{App}(n_1, n_2)]\!] := [\![r \overset{\tau \cdot \swarrow}{\rightsquigarrow} n_1]\!]\ [\![r \overset{\tau \cdot \searrow}{\rightsquigarrow} n_2]\!]$.

When $\tau = \epsilon$, we just write $[\![r]\!]$ instead of $[\![r \overset{\epsilon}{\rightsquigarrow}]\!]$.

Some remarks about Theorem 9:

- The hypothesis that r is a root node is necessary to ensure that the readback $[\![r \overset{\tau}{\rightsquigarrow}]\!]$ is a valid locally-nameless term. In fact Point 1 of the definition uses $\mathtt{index}(l \mid r \overset{\tau}{\rightsquigarrow})$, which is well-defined only if $r \overset{\tau}{\rightsquigarrow} \mathtt{Var}(l)$ crosses l. When r is a root node, this is the case by domination (Theorem 5).
- The definition is recursive, but it is not immediately clear what is the measure of termination. In fact, the readback calls itself recursively on longer paths. Still, the definition is well-posed because paths do not use binding edges, and because λ-graphs are finite and acyclic (Theorem 6).

4 THE THEORY OF SHARING EQUALITY

Sharing equivalence. To formalize the idea that two different λ-graphs unfold to the same λ-term, we introduce a general notion of equivalence between nodes (sharing equivalence, Theorem 11) whose intended meaning is that two related nodes have the same readback. This notion shall rest on various requirements; the first one is that only *homogeneous* nodes can be related:

Definition 10 (Homogeneous nodes [28]). *Let n, m be nodes of a λ-graph G. We say that n and m are* homogeneous *if they are*

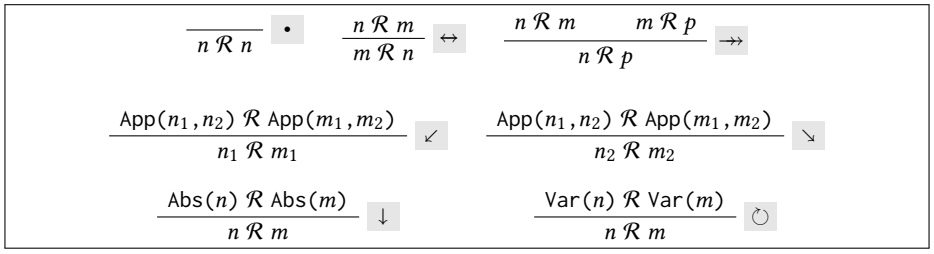


Figure 3: Sharing equivalence rules on a λ-graph

both application nodes, or they are both abstraction nodes, or they are both free variable nodes, or they are both bound variable nodes.

In the following, let us denote by $\mathcal{R}$ a generic binary relation over the nodes of a λ-graph G. We call a relation $\mathcal{R}$ *homogeneous* if it only relates pairs of homogeneous nodes, *i.e.* $n\ \mathcal{R}\ m$ implies that n and m are homogeneous.

Another requirement for sharing equivalences is that they must be closed under certain structural rules. More in general, in the rest of this paper we are going to characterize various kinds of relations as being closed under different sets of rules, which can all be found in Figure 3:

- *Equivalence rules*: rules • ↔ ↠ are the usual rules that characterize equivalence relations, respectively reflexivity, symmetry, and transitivity.
- *Bisimulation rules*:
  - *Downward propagation rules:* rules ↙ ↓ ↘ are downward propagation rules on the λ-graph. The rules ↙ and ↘ state that if two application nodes are related, then also their corresponding left and right children should be related. The rule ↓ states that if two abstraction nodes are related, then also their bodies should be related.
  - *Scoping rule:* the rule ↻ states that if two bound variable nodes are related, then also their binders should be related.

The last requirement for a sharing equivalence is the handling of free variable nodes: a sharing equivalence shall not equate two different free variable nodes, cf. the requirement *Open* in the upcoming definition.

We are now ready to define sharing equivalences:

Definition 11 ((Blind) sharing equivalences). *Let $\equiv$ be a binary relation over the nodes of a λ-graph G.*

- $\equiv$ *is a* blind sharing equivalence *if:*
  - Equivalence*: $\equiv$ is an equivalence relation;*
  - Blind bisimulation*: $\equiv$ is homogeneous and closed under the rules ↙ ↓ ↘ of Figure 3.*
- $\equiv$ *is a* sharing equivalence *if:*
  - Equivalence*: $\equiv$ is an equivalence relation;*
  - Bisimulation*: $\equiv$ is homogeneous and closed under the rules ↙ ↓ ↘ ↻ of Figure 3.*
  - Open*: $v \equiv w$ implies $v = w$ for every free variable nodes v, w.*

Equivalently, $\equiv$ is a sharing equivalence if it is a blind sharing equivalence and it also satisfies the following conditions on variable nodes: the open *requirement for free variable nodes, and the closure under ↻ for bound variable nodes.*

Example. Consider Fig. 2(a). The green (horizontal) waves are an economical representation of a sharing equivalence—nodes in the same class are connected by a green path, and reflexive/transitive waves are omitted.

Remark 12. *(Blind) sharing equivalences are closed by intersection, so that if there exists a (blind) sharing equivalence on a λ-graph then there is a smallest one.*

The requirements for a sharing equivalence $\equiv$ on a λ-graph G essentially ensure that G quotiented by $\equiv$ has itself the structure of a λ-graph. Note that blind sharing equivalences are not enough, because without the scoping rule, binders are not unique up to $\equiv$—it is nonetheless possible to prove that paths up to $\equiv$ are acyclic, which is going to be one of the key properties to prove the completeness of the Blind check.

Remark 13. *Let $\equiv$ be a blind sharing equivalence on a λ-graph G. Then:*

1. Acyclicity up to $\equiv$*: paths upto $\equiv$ in G are acyclic.*
2. Sharing equivalences as λ-graphs*: if $\equiv$ also satisfies the name conditions then $G/\equiv$ is a λ-graph.*

For instance, Fig. 2(b) shows the λ-graph corresponding to the quotient of the one of Fig. 2(a) by the sharing equivalence induced by the green waves.

Sharing equivalences do capture equality of readbacks, as we shall show, in the following sense (this is a sketch given to guide the reader towards the proper relationship, formalized by Theorem 18 at the end of this section):

- *sharing to α*: if $n \equiv m$ then $[\![n]\!] = [\![m]\!]$;
- *α to sharing*: if $[\![n]\!] = [\![m]\!]$ then there exists a sharing equivalence $\equiv$ such that $n \equiv m$.

(Spreaded) queries. According to the sketch we just provided, to check the sharing equality of two terms with sharing, *i.e.* two λ-graphs with roots n and m, it is enough to compute the smallest sharing equivalence $\equiv$ such that $n \equiv m$, if it exists, and failing otherwise. This is what our algorithm does. At the same time, however, it is slightly more general: it may test more than two nodes at the same time, *i.e.* all the pairs of nodes contained in a *query*.

Definition 14 (Query). *A query Q over a λ-graph G is a binary relation over the root nodes of G.*

The simplest case is when there are only two roots n and m and the query contains only $n\ Q\ m$ (depicted as a blue wave in Fig. 2(a))—from now on however we work with a generic query Q, which may relate nodes that are roots of not necessarily disjoint or even distinct λ-graphs; our focus is on the smallest sharing equivalence containing Q.

Let us be more precise. Every query Q induces a number of other *equality requests* obtained by closing Q with respect to the equivalence and propagation clauses that every sharing equivalence has to satisfy. In other words, every query induces a spreaded query.

Definition 15 (Spreading $\mathcal{R}^\#$). *Let $\mathcal{R}$ be a binary relation over the nodes of a λ-graph G. The spreading $\mathcal{R}^\#$ induced by $\mathcal{R}$ is the binary relation on the nodes of G inductively defined by closing $\mathcal{R}$ under the rules* $\bullet$ $\leftrightarrow$ $\twoheadrightarrow$ $\swarrow$ $\downarrow$ $\searrow$ *of Figure 3.*

Example. The spreading $Q^\#$ of the (blue) query Q in Fig. 2(a) is the (reflexive and transitive closure) of the green waves.

Blind universality of $Q^\#$. Note that the spreaded query $Q^\#$ is defined without knowing if there exists a (blind) sharing equivalence containing the query Q—there might very well be none (if the nodes are not sharing equivalent). It turns out that the spreaded query $Q^\#$ is itself a blind sharing equivalence, whenever there exists a blind sharing equivalence containing the query Q. In that case, unsurprisingly, $Q^\#$ is also the *smallest* blind sharing equivalence containing the query Q.

Proposition 16 (Blind universality of $Q^\#$). *Let Q be a query. If there exists a blind sharing equivalence $\equiv$ containing Q then:*

1. *The spreaded query $Q^\#$ is contained in $\equiv$, i.e. $Q^\# \subseteq \equiv$.*
2. *$Q^\#$ is the smallest blind sharing equivalence containing Q.*

Cycles up to $Q^\#$. Let us apply Theorem 13(1) to $Q^\#$, and take the contrapositive statement: if paths up to $Q^\#$ are cyclic then $Q^\#$ is not a blind sharing equivalence. The Blind Check in Sect. 6 indeed fails as soon as it finds a cycle up to $Q^\#$. Note, now, that $Q^\#$ satisfies the *equivalence* and *bisimulation* requirements for a blind sharing equivalence *by definition*. The only way in which it might not be such an equivalence then, is if it is not homogeneous. Said differently, there is in principle no need to check for cycles, it is enough to test for homogeneity. We are going to do it anyway, because cycles provide earlier failures—there are also other practical reasons to do so, to be discussed in Sect. 6.

Universality of the spreaded query $Q^\#$. Here it lies the key conceptual point in extending the linearity of Paterson and Wegman's algorithm to binders and working up to α-equivalence of bound variables.

In general the spreading $\mathcal{R}^\#$ of a binary relation $\mathcal{R}$ may not be a sharing equivalence, even though a sharing equivalence containing $\mathcal{R}$ exists. Consider for instance the relation $\mathcal{R}$ in Fig. 2(d): $\mathcal{R}$ coincides with its spreading $\mathcal{R}^\#$ (up to reflexivity), which is not a sharing equivalence because it does not include the `Abs`-nodes *above* the original query—note that spreading only happens downwards. To obtain a sharing equivalence one has to also include the `Abs`-nodes (Fig. 2(e)). The example does not show it, but in general then one has to start over spreading the new relation (eventually having to add other `Abs`-nodes found in the process, and so on). These iterations are obviously problematic in order to keep linear the complexity of the procedure—a key point of Paterson and Wegman's algorithm is that every node is processed only once.

What makes possible to extend their algorithm to binders is that the query is *context-free*, *i.e.* it is not in the scope of any abstraction, which is the case since it involves only pairs of roots, that therefore are above all abstractions as in Fig. 2(c). Then—remarkably—there is no need to iterate the propagation of the query. Said differently, if the relation Q is context-free then $Q^\#$ is universal.

The structural property of λ-graphs guaranteeing the absence of iterations for context-free queries is *domination*. Domination asks that to reach a bound variable from outside its scope one necessarily needs to first pass through its binder. The intuition is that if one starts with a context-free query then there is no need to iterate because binders are necessarily visited before the variables while propagating the query downwards.

Let us stress, however, that it is not evident that domination is enough. In fact, domination is about *one* bound variable and *its only* binder. For sharing equivalence instead one deals with a *class* of equivalent variables and a *class* of binders—said differently, domination is given in a setting without queries, and is not obvious that it gets along well with them. The fact that domination on single binders is enough for spreaded queries to be universal requires indeed a non-trivial proof and it is a somewhat surprising fact.

Proposition 17 (Universality of $Q^\#$). *Let Q be a query over a λ-graph G. If there exists a sharing equivalence $\equiv$ containing Q, then the spreaded query $Q^\#$ is the smallest sharing equivalence containing Q.*

The proof of this proposition is in [18, Appendix A], where it is obtained as a corollary of other results connecting equality of λ-terms and sharing equivalences, that also rely crucially on the the fact that queries only relate roots. It can also be proved directly, but it requires a very similar reasoning, which is why we rather prove it indirectly.

The sharing equality theorem. We have now introduced all the needed concepts to state the precise connection between the equality of λ-terms, queries, and sharing equivalences, which is the main result of our abstract study of sharing equality.

Theorem 18 (Sharing equality). *Let Q be a query over a λ-graph G. $Q^\#$ is a sharing equivalence if and only if $[\![n]\!] = [\![m]\!]$ for every nodes n, m such that $n\ Q\ m$.*

Despite the—we hope—quite intuitive nature of the theorem, its proof is delicate and requires a number of further concepts and lemmas, developed in [18, Appendices A–B]. The key point is finding an invariant expressing how queries on roots propagate under abstractions and interact with domination, until they eventually satisfy the name conditions for a sharing equivalence, and vice versa.

Let us conclude the section by stressing a subtlety of Theorem 18. Consider Fig. 2(c)—with that query the statement is satisfied. Consider Fig. 2(d)—that relation is not a valid query because it does not relate root nodes, and in fact the statement would fail because the readback of the two queried nodes are not the same and not even well-defined. Consider the relation $\mathcal{R}$ in Fig. 2(e)—now $\mathcal{R}$ and $\mathcal{R}^\#$ coincide (up to reflexivity) and $\mathcal{R}^\#$ is a sharing equivalence, but the

theorem (correctly) fails, because not all queried pairs of nodes are equivalent, as in Fig. 2(d).

5 ALGORITHMS FOR SHARING EQUALITY

From now on, we focus on the algorithmic side of sharing equality.

By the universality of spreaded queries $Q^\#$ (Proposition 17), checking the satisfability of a query Q boils down to compute $Q^\#$, and then check that it is a sharing equivalence: the fact that the requirements on variables are modular to the blind sharing requirement is one of our main contributions. Indeed it is possible to check sharing equality in two phases:

(1) *Blind Check*: building $Q^\#$ and at the same time checking that it is a blind sharing equivalence, *i.e.* that it is homogeneous;
(2) *Variables Check*: verifing that $Q^\#$ is a sharing equivalence by checking the conditions for free and bound variable nodes.

Of course, the difficulty is doing it in linear time, and it essentially lies in the Blind Check.

The rest of this part presents two algorithms, the Blind Check (Algorithm 1) and the Variables Check (Algorithm 2), with proofs of correctness and completeness, and complexity analyses. The second algorithm is actually straightforward. Be careful, however: the algorithm for the Variables Check is trivial just because the subtleties of this part have been isolated in the previous section.

6 THE BLIND CHECK

In this section we introduce the basic concepts for the Blind Check, plus the algorithm itself.

Our algorithm is a simple adaptation of Paterson and Wegman's, and it relies on the same key ideas in order to be linear. With respect to PW original algorithm, our reformulation does not rely on their notions of *dead/alive nodes* used to keep track of the nodes already processed; in addition it is not destructive, *i.e.* it does not remove edges and nodes from the graph, hence it is more suitable for use in computer tools where the λ-terms to be checked for equality need not be destroyed. Another contribution in this part is a formal proof of correctness and completeness, obtained via the isolation of properties of program runs.

Intuitions for the Blind Check. Paterson and Wegman's algorithm is based on a tricky, linear time visit of the λ-graph. It addresses two main efficiency issues:

(1) *The spreaded query is quadratic*: the number of pairs in the spreaded query $Q^\#$ can be quadratic in the size of the λ-graph. An equivalence class of cardinality n has indeed $\Omega(n^2)$ pairs for the relation—this is true for every equivalence relation. This point is addressed by rather computing a linear relation $=_c$ *generating* $Q^\#$, based on keeping a *canonical element* for every sharing equivalence class.
(2) *Merging equivalence classes*: merging equivalence classes is an operation that, for as efficient as it may be, it is not a costant time operation. The trickiness of the visit of the λ-graph is indeed meant to guarantee that, if the query is satisfiable, one never needs to merge two equivalence classes, but only to add single elements to classes.

More specifically, the ideas behind Algorithm 1 are:

Algorithm 1: The `BlindCheck` Algorithm

Data: an initial state
Result: *Fail* or a final state

```
1  Procedure BlindCheck()
2      for every node n do
3          if canonic(n) undefined then
             BuildEquivalenceClass(n)
4  Procedure BuildEquivalenceClass(c)
5      canonic(c) := c
6      building(c) := true
7      queue(c) := {c}
8      while queue(c) is non-empty do
9          n := queue(c).pop()
10         for every parent m of n do
11             case canonic(m) of
12                 undefined ⇒ BuildEquivalenceClass(m)
13                 c' ⇒ if building(c') then fail
14         for every ~neighbour m of n do
15             case canonic(m) of
16                 undefined ⇒ EnqueueAndPropagate(m, c)
17                 c' ⇒ if c' ≠ c then fail
18     building(c) := false
19 Procedure EnqueueAndPropagate(m, c)
20     case m , c of
21         Abs(m') , Abs(c') ⇒ create edge m' ~ c'
22         App(m1 , m2) , App(c1 , c2) ⇒
23             create edges m1 ~ c1 and m2 ~ c2
24         Var(l) , Var(l') ⇒ ()
25         Var() , Var() ⇒ ()
26         _ , _ ⇒ fail
27     canonic(m) := c
28     queue(c).push(m)
```

- *Top-down recursive exploration*: the algorithm can start on any node, not necessarily a root. However, when processing a node n the algorithm first makes a recursive call on the parents of n that have not been visited yet. This is done to avoid the risk of reprocessing n later because of some new equality requests on n coming from a parent processed after n.
- *Query edges*: the query is represented through additional *undirected query edges* between nodes, and it is propagated on child nodes by adding further query edges. The query is propagated carefully, on-demand. The fully propagated query is never computed, because, as explained, in general its size is quadratic in the number of nodes.
- *Canonic edges*: when a node is visited, it is assigned a canonic node that is a representative of its sharing equivalence class. This is represented via a *directed canonic edge*, which is implemented as a pointer.

- *Building flag*: each node has a boolean "building" flag that is used only by canonic representatives and notes the state of construction of their equivalence class. When undefined, it means that that node is not currently designated as canonic; when true, it means that the equivalence class of that canonic is still being computed; when false, it signals that the equivalence class has been completely computed.
- *Failures and cycles*: the algorithm fails in three cases. First, when it finds two nodes in the same class that are not homogeneous (Line 26), because then the approximation of $Q^{\#}$ that it is computing cannot be a blind sharing equivalence. The two other cases (on Line 13 and Line 17) the algorithm uses the fact that the canonic edge is already present to infer that it found a cycle up to $Q^{\#}$, and so, again $Q^{\#}$ cannot be a blind sharing equivalence (please read again the paragraph after Proposition 16).
- *Building a class*: calling `BuildEquivalenceClass`(n) boils down to
  (1) collect without duplicates all the nodes in the intended sharing equivalence class of n, that is, the nodes related to n by a sequence of query edges. This is done by the *while* loop at Line 14, that first collects the nodes queried with n and then iterates on the nodes queried with them. These nodes are inserted in a queue;
  (2) set n as the canonical element of its class, by setting the canonical edge of every node in the class (including n) to n;
  (3) propagate the query on the children (in case n is a `Lam` or a `App` node), by adding query edges between the corresponding children of every node in the class and their canonic.
  (4) Pushing a node in the queue, setting its canonic, and propagating the query on its children is done by the procedure `EnqueueAndPropagate`.
- *Linearity*: let us now come back to the two efficiency issues we mentioned before:
  - *Merging classes*: the recursive calls are done in order to guarantee that when a node is processed all the query edges for its sharing class are already available, so that the class shall not be extended nor merged with other classes later on during the visit of the λ-graph.
  - *Propagating the query*: the query is propagated only *after* having set the canonics of the current sharing equivalence class. To explain, consider a class of k nodes, which in general can be defined by $\Omega(k^2)$ query edges. Note that after canonization, the class is represented using only $k-1$ canonic edges, and thus the algorithm propagates only $O(k)$ query edges—this is why the number of query edges is kept linear in the number of the nodes (assuming that the original query itself was linear). If instead one would propagate query edges *before* canonizing the class, then the number of query edges may grow quadratically.

States. As explained, the algorithm needs to enrich λ-graphs with a few additional concepts, namely *canonic edges*, *query edges*, *building flags*, *queues*, and *execution stack*, all grouped under the notion of *program state*.

A *state* $\mathcal{S}$ of the algorithm is either $\mathcal{F}ail$ or a tuple

$$(G, \texttt{undir}, \texttt{canonic}, \texttt{building}, \texttt{queue}, \texttt{active})$$

where G is a λ-graph, and the remaining data structures have the following properties:

- *Undirected query edges ($\sim$)*: `undir` is a *multiset* of undirected query edges, pairing nodes that are expected to be placed by the algorithm in the same sharing equivalence class. Undirected loops are admitted and there may be multiple occurrences of an undirected edge between two nodes. More precisely, for every undirected edge between n and m with multiplicty k in the state, both (n, m) and (m, n) belong with multiplicity k to `undir`. We denote by $\sim$ the binary relation over G such that $n \sim m$ iff the edge (n, m) belongs to `undir`.
- *Canonic edges (*`c`*)*: nodes may have one additional `canonic` directed edge pointing to the computed canonical representative of that node. The partial function mapping each node to its canonical representative (if defined) is noted `c`. We then write $\texttt{c}(n) = m$ if the canonical of n is m, and $\texttt{c}(n) = \texttt{undefined}$ otherwise. By abuse of notation, we also consider `c` a binary relation on G, where n `c` m iff $\texttt{c}(n) = m$.
- *Building flags (*`b`*)*: nodes may have an additional boolean flag `building` that signals whether an equivalence class has or has not been constructed yet. The partial function mapping each node to its building flag (if defined) is noted `b`. We then write $\texttt{b}(n) = \texttt{true} \mid \texttt{false}$ if the building flag of n is defined, and $\texttt{b}(n) = \texttt{undefined}$ otherwise.
- *Queues (*`q`*)*: nodes have a queue data structure that is used only on canonic representatives, and contains the nodes of the class that are going to be processed next. The partial function mapping each node to its queue (if defined) is noted `q`.
- *Active Calls (*`active`*)*: a program state contains information on the execution stack of the algorithm, including the active procedures, local variables, and current execution line. We leave the concept of execution stack informal; we only define more formally `active`, which records the order of visit of equivalence classes that are under construction, and that is essential in the proof of completeness of the algorithm. `active` is an abstraction of the implicit execution stack of active calls to the procedure `BuildEquivalenceClass` where only (part of) the activation frames for $\texttt{BuildEquivalenceClass}(c)$ are represented. Formally, `active` is simply a sequence of nodes of the λ-graph, and $\texttt{active} = [c_1, \ldots, c_K]$ if and only if:
  - *Active:* $\texttt{BuildEquivalenceClass}(c_1)$, …, $\texttt{BuildEquivalenceClass}(c_K)$ are exactly the calls to `BuildEquivalenceClass` that are currently active, *i.e.* have been called before $\mathcal{S}$, but have not yet returned;
  - *Call Order:* for every $0 < i < j \leq K$, $\texttt{BuildEquivalenceClass}(c_i)$ was called before $\texttt{BuildEquivalenceClass}(c_j)$.

Moreover we introduce the following concepts:

- *Program transition*: the change of state caused by the execution of a piece of code. For the sake of readability, we avoid a technical definition of transitions; roughly, a transition is the

execution of a line of code, as they appear numbered in the algorithm itself. When the line is a `while` loop, a transition is an iteration of the body, or the exit from the loop; when the line is a `if-then-else`, a transition is entering one of the branches according to the condition; and so on.
- *Fail state*: $\mathcal{Fail}$ is the state reached after executing a `fail` instruction; it has no attributes and no transitions.
- *Initial state* $\mathcal{S}_0$: a non-$\mathcal{Fail}$ state with the following attributes:
  - *Initial* $\sim$*edges:* simply the initial query, *i.e.* $\sim \coloneqq Q$.
  - *Initial assignments:* $\mathsf{c}(n)$, $\mathsf{b}(n)$, and $\mathsf{q}(n)$ are undefined for every node n of G.
  - *Initial transition:* the first transition is a call to `BlindCheck()`.
- *Program run*: a sequence of program states starting from $\mathcal{S}_0$ obtained by consecutive transitions.
- *Reachable*: a state which is the last state of a program run.
- *Failing*: a reachable state that transitions to $\mathcal{Fail}$.
- *Final*: a non-$\mathcal{Fail}$ reachable state that has no further transitions.

Details about how the additional structures of enriched λ-graphs are implemented are given in Sect. 6.2, where the complexity of the algorithm is analysed.

6.1 Correctness and Completeness

Here we prove that Algorithm 1 correctly and completely solves the blind sharing equivalence problem, that is, it checks whether the spreaded query is a blind sharing equivalence.

The proofs rely on a number of general properties of program runs and reachable states. The full statements and proofs are in [18, Appendix D], grouped according to the concepts that they analyze. Clearly the first step is showing that the mechanism of equivalence classes is correct, *i.e.* that for every node n and every program run:

- $\mathsf{canonic}(n)$ is assigned at most once;
- $\mathsf{canonic}(n)$ is never assigned to undefined;
- if $\mathsf{canonic}(n)$ is assigned to a node c, then $\mathsf{canonic}(c)$ is assigned to c itself.

The first two points guarantee that nodes never change equivalence class, *i.e.* that when a node is assigned to an equivalence class, it stays so for the rest of the execution of the algorithm; the third point ensures that a canonic node is actually a representative of its own class. Other important properties concern data structures used by the algorithm—for example, queues and the operations of enqueing/dequeuing—or loops—like the visit of parents and $\sim$neighbours of a node. With respect to queues, it is relevant to note that a node is never enqued twice (fact useful as well when proving the linear-time complexity of the algorithm), and that the queue of a canonic c only contains nodes of canonic assigned to c.

The loop on Line 10 visits the parents of a node n by calling `BuildEquivalenceClass` on each parent. After the execution of the loop:

- *Parent classes built:* all equivalence classes of the parents of n are completely built;
- *Finalization of* $\sim$*neighbours:* the set of $\sim$neighbours of n is not going to change during the program run.

Finally, the loop on Line 14 visits the $\sim$neighbours of a node n, and for every sibling m it enforces that n and m have the same canonic assigned: hence after the loop all query edges with endnode n are subsumed by the canonic assignment.

Correctness. We prove that whenever Algorithm 1 terminates successfully with final state $\mathcal{S}_f$ from an initial query Q, then $Q^\#$ is homogeneous, and therefore a blind sharing equivalence. Additionally, we show that the canonic assignment is a succint representation of $Q^\#$, *i.e.* that for all nodes n, m: $n\ Q^\#\ m$ if and only if n and m have the same canonic assigned in $\mathcal{S}_f$. The following are the most important properties to prove correctness:

- *Blind bisimulation upto:* in all reachable states, c is a blind bisimulation upto $(\sim \cup =)$.
- *Undirected query edges approximate the spreaded query:* in all reachable states, $Q \subseteq \sim \subseteq Q^\#$.
- *The canonic assignment respects the* $\sim$ *constraints:* in all reachable states, $\mathsf{c} \subseteq \sim^*$
- *Eventually, all query edges are visited:* in all final states, all query edges are subsumed by the canonic assignment, *i.e.* $\sim \subseteq \mathsf{c}^*$.

In every reachable state, we define $=_\mathsf{c}$ as the equivalence relation over the λ-graph that equates two nodes whenever their canonic representatives are both defined and coincide. The lemmas above basically state that during the execution of the algorithm, $=_\mathsf{c}$ is a blind sharing equivalence up to the query edges, and that $\sim$ can indeed be seen as an approximation of the spreaded query $Q^\#$. At the end of the algorithm, $\sim$ and c actually represent the same (up to the $(\cdot)^*$ closure) relation $=_\mathsf{c}$, which is then exactly the spreaded query:

Proposition 19 (Correctness). *In every final state* $\mathcal{S}_f$*:*

- Succint representation: $(=_\mathsf{c}) = (Q^\#)$,
- Blind check: $Q^\#$ *is homogeneous, and therefore a blind sharing equivalence.*

Completeness. Completeness is the fact that whenever Algorithm 1 fails, $Q^\#$ is not a blind sharing equivalence. Recall that the algorithm can fail only while executing the following three lines of code:

- On Line 13 during `BuildEquivalenceClass`, when a node n is being processed and it has a parent whose equivalence class is still being built;
- On Line 17 during `BuildEquivalenceClass`, when a node n is being processed and it has a $\sim$neighbour belonging to a difference equivalence class;
- On Line 26 during `EnqueueAndPropagate(`m,c`)`, when the algorithm is trying two relate the nodes m and c which are not homogeneous.

While in the latter case (Line 26) the failure of the homogeneous condition is more evident, in the first two cases it is more subtle. In fact, when the homogeneous condition fails on Line 13 and Line 17 it is not because we explicitly found two related nodes that are not homogeneous, but because $Q^\#$ does not satisfy an indirect property that is necessary for $Q^\#$ to be homogenous (see below). In these cases the algorithm fails early, even though it has not visited yet the actual pair of nodes that are not homogeneous.

To justify the early failures we use `active`, which basically records the equivalence classes that the algorithm is building in a given state, sorted according to the order of calls to `BuildEquivalenceClass`. Nodes in `active` respect a certain strict order: if `active` = $[c_1, \dots, c_K]$, then $c_1 \prec c_2 \prec \dots \prec c_K$ (for the formal definition of $\prec$ see [18, Appendix D]), where $n \prec m$ implies that the equivalence class of n is a child of the one of m in the quotient graph. The algorithm fails on Line 13 and Line 17 because it found a cyclic chain of nodes related by $\prec$, therefore finding a cycle in the quotient graph. By Theorem 13, there cannot exist any blind sharing equivalence containing the initial query in this case.

Proposition 20 (Completeness). *If Algorithm 1 fails, then $\mathcal{Q}^\#$ is not a blind sharing equivalence, and therefore by the blind universality of $\mathcal{Q}^\#$ (Theorem 16), there does not exist any blind sharing equivalence containing the initial query $\mathcal{Q}$.*

6.2 Linearity

In this section we show that Algorithm 1 always terminates, and it does so in time linear in the size of the λ-graph and the initial query.

Low-level assumptions. In order to analyse the complexity of the algorithm we have to spell out some details about an hypothetical implementation on a RAM of the data structures used by the Blind check.

- *Nodes*: since Line 2 of the algorithm needs to iterate on all nodes of the λ-graph, we assume an array of pointers to all the nodes of the graph.
- *Directed edges*: these edges—despite being directed—have to be traversed in both directions, typically to recurse over the parents of a node. We then assume that every node has an array of pointers to its parents.
- *Undirected query edges*: query edges are undirected and are dynamically created; in addition, the algorithm needs to iterate on all ∼neighbours of a node. In order to obtain the right complexity, every node simply maintains a linked list of its ∼neighbours, in such a way that when a new undirected edge (n, m) is created, then n is pushed on the list of ∼neighbours of m, and m on the one of n.
- *Canonical assignment* is obtained by a pointer to a node (possibly `undefined`) in the data structure for nodes.
- *Building flags* are just implemented via a boolean on each node.
- *Queues* do not need to be recorded in the data structure for nodes, as they can be equivalently coded as local variables.

Let us call *atomic* the following operations performed by the check: finding the first node, finding the next node given the previous node, finding the first parent of a node, finding the next parent of a node given the previous parent, checking and setting canonics, checking and setting building flags, getting the next query edge on a given node, traversing a query edge, adding a query edge between two nodes, pushing to a queue, and popping an element off of a queue.

Lemma 21 (Atomic operations are constant). *The atomic operations of the Blind check are all implementable in constant time on a RAM.*

We prove termination and linearity of Algorithm 1 via a global estimation of the number of transitions in a program run. The difficult part is to estimate the number of transitions executing lines of `BuildEquivalenceClass`, since it contains multiple nested loops. First of all, we note that in every program run `BuildEquivalenceClass` is called at most once for each node. Also the body of the while loop on Line 8 is executed in total at most once for each node, because in every program run each node is enqueued at most once. The loop on Line 10 is not problematic because the parents of a node do not change during a program run. Estimating the number of iterations of the loop on Line 14, which iterates over the ∼neighbours of a node n, seems much more involved because the code inside the loop may create new query edges; however as already discussed the ∼neighbours of n do not change after the parents of n are visited on Line 10. The last insight for linearity is to recall that the algorithm parsimoniously propagates query edges only between a node and its canonic: as a consequence, in every reachable state, `|undir|` is linear in the size of $\mathcal{Q}$ and the number of nodes in the λ-graph:

Proposition 22 (Bound on `undir`). *Let $\mathcal{Q}$ be a query over a λ-graph G as in input to Algorithm 1. In every reachable state $\mathcal{S}$, $|\texttt{undir}| \leq 2 \times |\mathcal{Q}| + 4 \times |N|$ (where $|N|$ is the number of nodes of G).*

Let $|G|$ denote the size of a λ-graph G, *i.e.* the number of nodes and edges of G. Then:

Proposition 23 (Linear-time termination). *Let $\mathcal{S}_0$ be an initial state of the algorithm, with a query $\mathcal{Q}$ over a λ-graph G. Then the Blind check terminates in a number of transitions linear in $|G|$ and $|\mathcal{Q}|$.*

7 THE VARIABLES CHECK

Our second algorithm (Algorithm 2) takes in input the output of the Blind check, that is, a blind sharing equivalence on a λ-graph G represented via canonic edges, and checks whether the `Var`-nodes of G satisfy the variables conditions for a sharing equivalence—free variable nodes at line 7, and bound variable nodes at line 9.

The Variables check is based on the fact that to compare a node with all those in its class it is enough to compare it with the canonical representative of the class—note that this fact is used twice, for the `Var`-nodes and for their binders. The check fails in two cases, either when checking a free variable node (in this case $\mathcal{Q}^\#$ is not an open relation) or when checking a bound variable node (in this case $\mathcal{Q}^\#$ is not closed under $\circlearrowleft$).

Theorem 24 (Correctness & completeness of the Variables check).
Let $\mathcal{Q}$ be a query over a λ-graph G passing the Blind check, and let c *be the canonic assignment produced by that check.*

- (Completeness) *If the Variables check fails then there are no sharing equivalences containing $\mathcal{Q}$,*
- (Correctness) *otherwise $=_{\texttt{c}}$ is the smallest sharing equivalence containing $\mathcal{Q}$.*

Moreover, the Variables check clearly terminates in time linear in the size of G.

Composing Theorem 19, Theorem 20, Theorem 23, and Theorem 24, we obtain the second main result of the paper.

Algorithm 2: Variables Check

Data: canonic(·) representation of $Q^{\#}$
Result: is $Q^{\#}$ a sharing equivalence?

```
1  Procedure VarCheck()
3      foreach Var-node v do
4          w ← canonic(v)
5          if v ≠ w then
7              if binder(v) or binder(w) is undefined
8               then fail                  // free Var-nodes
9              else if
                canonic(binder(v)) ≠ canonic(binder(w))
               then fail                  // bound Var-nodes
10         end
11     end
```

Theorem 25 (Sharing equality is linear). *Let Q be a query over a λ-graph G. The sharing equality algorithm obtained by combining Algorithm 1 and Algorithm 2 runs in time linear in the sizes of G and Q, and it succeeds if and only if there exists a sharing equivalence containing Q. Moreover, if it succeeds, it outputs a concrete (and linear) representation of the smallest such sharing equivalence.*

Finally, composing with the sharing equality theorem (Theorem 18) one obtains that the algorithm indeed tests the equality of the readbacks of the query, as expected.

8 CONCLUSIONS

We presented the first linear-time algorithm to check the sharing equivalence of λ-graphs. Following our development of the theory of sharing equality, we split the algorithm in two parts: a first part performing a check on the DAG structure of λ-graphs, and the second part checking the additional requirements on variables and scopes. The paper is accompanied by a technical report with formal proofs of correctness, completeness, and termination in linear time.

The motivation for this work stemmed from our previous works on the evaluation of λ-terms: abstract machines can reduce a λ-term to normal form in time bilinear in the size of the term and the number of evaluation steps, if λ-terms are represented in memory as λ-graphs. When combining bilinear evaluation through abstract machines and the sharing equality algorithm presented here, one obtains a bilinear algorithm for conversion. As a consequence, $\alpha\beta$-conversion can be implemented with only bilinear overhead.

ACKNOWLEDGMENTS

This work has been partially funded by the ANR JCJC grant COCA HOLA (ANR-16-CE40-004-01) and the COST Action EUTypes CA15123 STSM#43345.

REFERENCES

[1] Beniamino Accattoli. 2018. Proof Nets and the Linear Substitution Calculus. In *Theoretical Aspects of Computing - ICTAC 2018 - 15th International Colloquium, Stellenbosch, South Africa, October 16-19, 2018, Proceedings*. 37–61. https://doi.org/10.1007/978-3-030-02508-3_3

[2] Beniamino Accattoli and Bruno Barras. 2017. Environments and the complexity of abstract machines. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, Namur, Belgium, October 09 - 11, 2017*. 4–16. https://doi.org/10.1145/3131851.3131855

[3] Beniamino Accattoli and Claudio Sacerdoti Coen. 2015. On the Relative Usefulness of Fireballs. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*. 141–155. https://doi.org/10.1109/LICS.2015.23

[4] Beniamino Accattoli and Giulio Guerrieri. 2017. Implementing Open Call-by-Value. In *Fundamentals of Software Engineering - 7th International Conference, FSEN 2017, Tehran, Iran, April 26-28, 2017, Revised Selected Papers*. 1–19. https://doi.org/10.1007/978-3-319-68972-2_1

[5] Beniamino Accattoli and Ugo Dal Lago. 2012. On the Invariance of the Unitary Cost Model for Head Reduction. In *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*. 22–37. https://doi.org/10.4230/LIPIcs.RTA.2012.22

[6] Beniamino Accattoli and Ugo Dal Lago. 2014. Beta reduction is invariant, indeed. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*. 8:1–8:10. https://doi.org/10.1145/2603088.2603105

[7] John Allen. 1978. *Anatomy of LISP*. McGraw-Hill, Inc., New York, NY, USA.

[8] Stephen Alstrup, Dov Harel, Peter W. Lauridsen, and Mikkel Thorup. 1999. Dominators in Linear Time. *SIAM J. Comput.* 28, 6 (1999), 2117–2132. https://doi.org/10.1137/S0097539797317263

[9] Zena M. Ariola and Jan Willem Klop. 1996. Equational Term Graph Rewriting. *Fundam. Inform.* 26, 3/4 (1996), 207–240. https://doi.org/10.3233/FI-1996-263401

[10] Guy E. Blelloch and John Greiner. 1995. Parallelism in Sequential Functional Languages. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*. 226–237. https://doi.org/10.1145/224164.224210

[11] Stefanus Cornelis Christoffel Blom. 2001. *Term Graph Rewriting. Syntax and semantics*. Ph.D. Dissertation. Vrije Universiteit Amsterdam.

[12] Robert S Boyer and Jay S Moore. 1972. The sharing of structure in theorem-proving programs. *Machine intelligence* 7 (1972), 101–116.

[13] Adam L. Buchsbaum, Haim Kaplan, Anne Rogers, and Jeffery R. Westbrook. 1998. A New, Simpler Linear-Time Dominators Algorithm. *ACM Trans. Program. Lang. Syst.* 20, 6 (1998), 1265–1296. https://doi.org/10.1145/295656.295663

[14] Christophe Calvès. 2013. Unifying Nominal Unification. In *24th International Conference on Rewriting Techniques and Applications, RTA 2013, June 24-26, 2013, Eindhoven, The Netherlands*. 143–157. https://doi.org/10.4230/LIPIcs.RTA.2013.143

[15] Christophe Calvès and Maribel Fernández. 2010. The First-Order Nominal Link. In *Logic-Based Program Synthesis and Transformation - 20th International Symposium, LOPSTR 2010, Hagenberg, Austria, July 23-25, 2010, Revised Selected Papers*. 234–248. https://doi.org/10.1007/978-3-642-20551-4_15

[16] Christophe Calvès and Maribel Fernández. 2010. Matching and alpha-equivalence check for nominal terms. *J. Comput. Syst. Sci.* 76, 5 (2010), 283–301. https://doi.org/10.1016/j.jcss.2009.10.003

[17] Arthur Charguéraud. 2012. The Locally Nameless Representation. *J. Autom. Reasoning* 49, 3 (2012), 363–408. https://doi.org/10.1007/s10817-011-9225-2

[18] Andrea Condoluci, Beniamino Accattoli, and Claudio Sacerdoti Coen. 2019. Sharing Equality is Linear. *arXiv e-prints* (Jul 2019). arXiv:1907.06101

[19] Andrei P. Ershov. 1958. On Programming of Arithmetic Operations. *Commun. ACM* 1, 8 (1958), 3–9. https://doi.org/10.1145/368892.368907

[20] Harold N. Gabow. 1990. Data Structures for Weighted Matching and Nearest Common Ancestors with Linking. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California, USA*. 434–443. http://dl.acm.org/citation.cfm?id=320176.320229

[21] Eiichi Goto. 1974. *Monocopy and associative algorithms in extended Lisp*. Technical report TR 74–03. University of Tokyo.

[22] Clemens Grabmayer and Jan Rochel. 2014. Maximal sharing in the Lambda calculus with letrec. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*. 67–80. https://doi.org/10.1145/2628136.2628148

[23] J. Hopcroft and R. Karp. 1971. *A Linear Algorithm for Testing Equivalence of Finite Automata*. Technical Report 0. Dept. of Computer Science, Cornell U.

[24] John Lamping. 1990. An Algorithm for Optimal Lambda Calculus Reduction. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*. 16–30. https://doi.org/10.1145/96709.96711

[25] Jordi Levy and Mateu Villaret. 2010. An Efficient Nominal Unification Algorithm. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010, July 11-13, 2010, Edinburgh, Scottland, UK*. 209–226. https://doi.org/10.4230/LIPIcs.RTA.2010.209

[26] Alberto Martelli and Ugo Montanari. 1977. Theorem Proving with Structure Sharing and Efficient Unification. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence. Cambridge, MA, USA, August 22-25, 1977*. 543. http://ijcai.org/Proceedings/77-1/Papers/096.pdf

[27] Alberto Martelli and Ugo Montanari. 1982. An Efficient Unification Algorithm. *ACM Trans. Program. Lang. Syst.* 4, 2 (1982), 258–282. https://doi.org/10.1145/357162.357169

[28] M.S. Paterson and M.N. Wegman. 1978. Linear unification. *J. Comput. System Sci.* 16, 2 (1978), 158 – 167. https://doi.org/10.1016/0022-0000(78)90043-0

[29] Zhenyu Qian. 1993. Linear Unification of Higher-Order Patterns. In *TAPSOFT'93: Theory and Practice of Software Development, International Joint Conference CAAP/FASE, Orsay, France, April 13-17, 1993, Proceedings*. 391–405. https://doi.org/10.1007/3-540-56610-4_78

[30] Laurent Regnier. 1992. *Lambda-calcul et réseaux*. PhD thesis. Univ. Paris VII.

[31] Manfred Schmidt-Schauß, Conrad Rau, and David Sabel. 2013. Algorithms for Extended Alpha-Equivalence and Complexity. In *24th International Conference on Rewriting Techniques and Applications, RTA 2013, June 24-26, 2013, Eindhoven, The Netherlands*. 255–270. https://doi.org/10.4230/LIPIcs.RTA.2013.255

[32] Christian Urban, Andrew M. Pitts, and Murdoch Gabbay. 2003. Nominal Unificaiton. In *Computer Science Logic, 17th International Workshop, CSL 2003, 12th Annual Conference of the EACSL, and 8th Kurt Gödel Colloquium, KGC 2003, Vienna, Austria, August 25-30, 2003, Proceedings*. 513–527. https://doi.org/10.1007/978-3-540-45220-1_41

[33] C.P. Wadsworth. 1971. *Semantics and Pragmatics of the Lambda-calculus*. University of Oxford. https://books.google.it/books?id=kl1QIQAACAAJ